Name:- Shazim Khot
Roll no:-25AI57
Class:- SE AIML A
Batch:-3

Experiment No. 2 Title: Implementation of Stack

Stack:-

A Stack is a linear data structure that works on the LIFO (Last In, First Out) principle, where the most recently added element is the first one to be removed. Both insertion (push) and deletion (pop) operations take place only at one end of the stack, known as the top. Stacks are widely used in computer science for tasks such as managing function calls, evaluating expressions, maintaining browser history, and implementing undo/redo features in applications.

LIFO:-

LIFO (Last In, First Out) is the working principle of a stack, which means the latest element added is the first one to be removed. In other words, the last item inserted into the stack is the first item that comes out.
Example:
If the elements 10, 20, and 30 are pushed onto a stack, it will look like this:
30 ← Top
20
10
When a Pop operation is performed, the element 30 is removed first because it was the last element inserted into the stack.

Push Operation:-

The Push operation is used to add a new element to the top of the stack. Before the element is inserted, the system verifies whether the stack has reached its maximum capacity. If the stack is not full, the new element is placed on the top.

Pop Operation:-

The Pop operation is used to remove the element from the top of the stack. Before removing an element, the program checks whether the stack is empty. If there are no elements in the stack, the operation results in Stack Underflow.

Peek Operation:-

The Peek operation is used to view the element at the top of the stack without removing it. It allows you to access the topmost element while keeping the stack unchanged.

isEmpty():-

The IsEmpty function is used to determine whether the stack is empty. It returns true if the stack contains no elements; otherwise, it returns false.

isFull():-

The IsFull function is used to check whether the stack has reached its maximum storage capacity. It returns true if the stack cannot accommodate any more elements; otherwise, it returns false.

Experiment 2.A

Aim:-

Implement a Stack using arrays.

Theory:-

A stack can be implemented using an array by using a variable named top, which keeps track of the position of the most recently added element. Initially, the value of top is set to -1, indicating that the stack does not contain any elements. During a Push operation, the value of top is increased first, and then the new element is inserted. During a Pop operation, the element present at the top position is removed, and the value of top is decreased. Other operations such as Peek, IsEmpty, and IsFull are used to access and manage the stack effectively.

Code:-

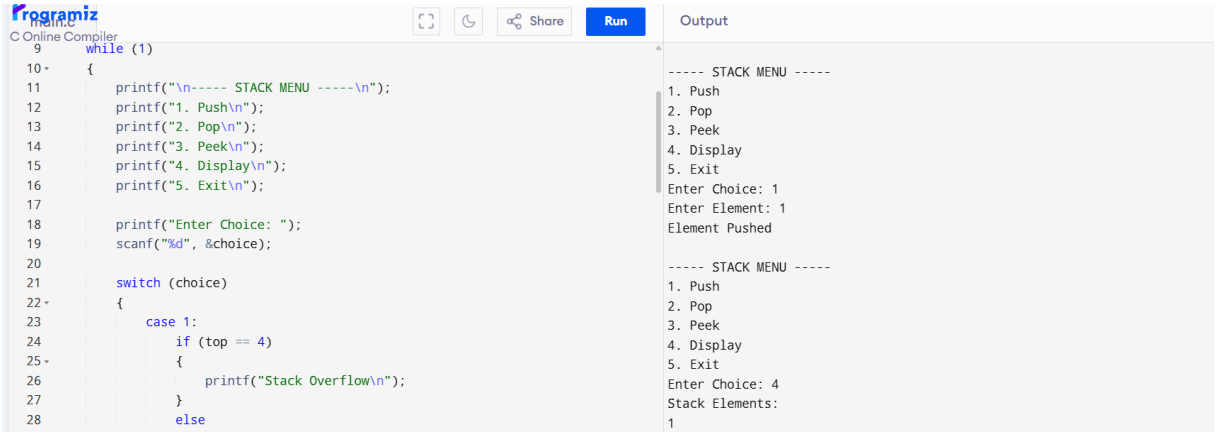
```
#include <stdio.h>

int main()
{
   int stack[5];
   int top = -1;
   int choice, value;
```

```
while (1)
{
   printf("\n----- STACK MENU -----\n");
   printf("1. Push\n");
   printf("2. Pop\n");
   printf("3. Peek\n");
   printf("4. Display\n");
   printf("5. Exit\n");

   printf("Enter Choice: ");
   scanf("%d", &choice);

   switch (choice)
   {
      case 1:
         if (top == 4)
         {
            printf("Stack Overflow\n");
         }
         else
         {
            printf("Enter Element: ");
            scanf("%d", &value);

            stack[++top] = value;
            printf("Element Pushed\n");
         }
         break;

      case 2:
         if (top == -1)
         {
            printf("Stack Underflow\n");
         }
         else
         {
```

```
            printf("Deleted Element: %d\n", stack[top--]);
        }
        break;

    case 3:
        if (top == -1)
        {
            printf("Stack is Empty\n");
        }
        else
        {
            printf("Top Element: %d\n", stack[top]);
        }
        break;

    case 4:
        if (top == -1)
        {
            printf("Stack is Empty\n");
        }
        else
        {
            printf("Stack Elements:\n");

            for (int i = top; i >= 0; i--)
            {
                printf("%d\n", stack[i]);
            }
        }
        break;

    case 5:
        printf("Program Ended\n");
        return 0;

    default:
        printf("Invalid Choice\n");
```

```
        }
    }

    return 0;
}
```

Output:-



Conclusion:-

The stack was successfully implemented using an array. The program performed various basic stack operations such as Push, Pop, Peek, and Display. This experiment provided a clear understanding of the LIFO (Last In, First Out) principle and demonstrated how arrays can be used to efficiently manage stack operations.

Experiment 2.B

Aim:

Implement Infix to postfix conversion.

What is Infix Expression?

An Infix Expression is the standard mathematical notation in which operators are written between operands.
Example:
A+B
A*B+C

What is Postfix Expression?

In a Postfix Expression, the operator is written after its operands. Parentheses are not required because the order of evaluation is already clear.
Example
AB+
AB*C+

Why Convert Infix to Postfix?

Postfix notation makes expression evaluation simpler for computers because it eliminates the requirement of using parentheses and checking operator precedence. This allows expressions to be evaluated directly and efficiently using stack operations.

Operator Precedence:-
Operator Priority
^ Highest
* / High
+ - Low

Theory:-

Infix expressions are simple and understandable for humans, but they are not easily processed by computers. To make evaluation easier, infix expressions are converted into postfix expressions using a stack. During this conversion, operands are directly placed in the output, while operators are stored in the stack based on their precedence and associativity. Parentheses are managed using stack operations to ensure the correct sequence of evaluation is maintained.

Code:-

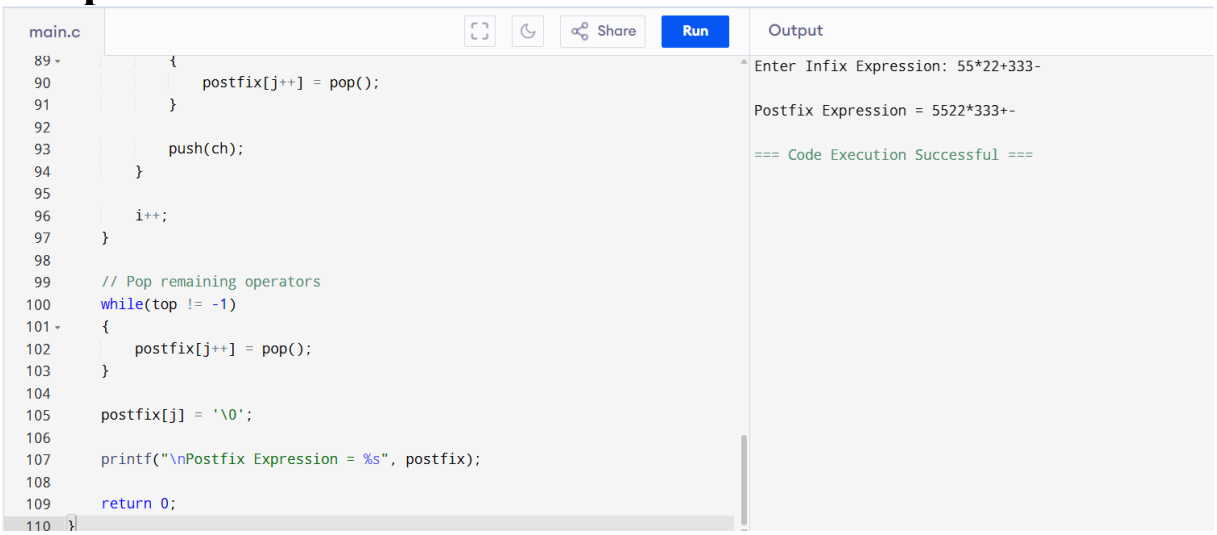
```
#include <stdio.h>
#include <ctype.h>

#define MAX 100

char stack[MAX];
int top = -1;

// Push operation
void push(char ch)
{
   stack[++top] = ch;
}

// Pop operation
char pop()
{
   return stack[top--];
}

// Function to check operator precedence
int precedence(char ch)
{
   switch(ch)
   {
      case '^':
         return 3;
```

```
        case '*':
        case '/':
        case '%':
            return 2;

        case '+':
        case '-':
            return 1;

        default:
            return 0;
    }
}

int main()
{
    char infix[MAX], postfix[MAX];
    int i = 0, j = 0;
    char ch;

    printf("Enter Infix Expression: ");
    scanf("%s", infix);

    while(infix[i] != '\0')
    {
        ch = infix[i];

        // If operand, add directly to postfix
        if(isalnum(ch))
        {
            postfix[j++] = ch;
        }

        // If opening bracket, push into stack
        else if(ch == '(' || ch == '{' || ch == '['
        {
```

```
        push(ch);
    }

    // If closing bracket, pop until opening bracket
    else if(ch == ')' || ch == '}' || ch == ']')
    {
        while(top != -1 &&
              stack[top] != '(' &&
              stack[top] != '{' &&
              stack[top] != '[')
        {
            postfix[j++] = pop();
        }

        // Remove opening bracket
        if(top != -1)
            pop();
    }

    // If operator
    else
    {
        while(top != -1 &&
              precedence(stack[top]) >= precedence(ch))
        {
            postfix[j++] = pop();
        }

        push(ch);
    }

    i++;
}

// Pop remaining operators
while(top != -1)
{
```

```
        postfix[j++] = pop();
    }

    postfix[j] = '\0';

    printf("\nPostfix Expression = %s", postfix);

    return 0;
}
```

Output:-



Conclusion:-

The infix to postfix conversion program was successfully implemented using the stack data structure. The experiment demonstrated how stacks help manage operators based on precedence and parentheses while converting expressions. It also provided a clear understanding of expression conversion techniques that are widely used in compilers and expression evaluation algorithms.